

# A `fork()` in the road: An Exploration

Chris Sinclair

Loyola University Chicago

---

## Abstract

The `fork()` system call in the Linux kernel is an artifact of the Unix philosophy of process creation that predates modern multithreading and memory management techniques. The 2019 publication that inspired this project, titled “A `fork()` in the road”, recommends deprecating the system call altogether and utilizing alternatives such as `vfork()` and `clone()` on a case-by-case basis. Unfortunately, despite its shortcomings, many modern systems still make use of `fork()`. Thus, the paper lands on a more pragmatic solution: educate *around* `fork()` when introducing the topic of process creation. This can help future developers understand the concept more thoroughly while still making them aware of the historical nature of `fork()`. It also allows for a “soft deprecation” by means of reducing the use of `fork()` in the long-term.

Interestingly, a significant point made in the paper revolves around the `CreateProcess()` API in Windows, which serves as a method that implements both `fork()` and its common counterpart, `exec()`, while also placing constraints on the resulting process that are relevant to modern software development techniques. This paper will detail my journey into the implementation of a system call in the Linux kernel that will essentially emulate the `CreateProcess()` function in Windows as a proper, modern alternative to `fork()`.

---

## 1. Background: `fork()` vs. `CreateProcess()`

The `fork()` system call creates a new child process that is a duplicate of the parent process, beginning right after the initial `fork()` call. Both processes share the same context and execution state, though they have their own separate process IDs. It is often accompanied by the `exec()` system call in the child process, which replaces its memory space with a new program by using the existing process ID but replacing it with new code and data.

On the other hand, `CreateProcess()` essentially combines the functionality of both `fork()` and `exec()` while adding extensive parameters to make process creation more robust. Some of these parameters include security attributes, priority settings, and environment variables. Rather than duplicating the parent process, the child process is actually just a new process entirely. By defining these parameters, a programmer has more control over how the new process is created without getting pulled into the mire of managing those aspects within the process itself.

## 2. Process

### 2.1 Scope

My personal familiarity with Linux has only extended to some basic high school tinkering with Red Hat Fedora Core 2 and then installing Ubuntu on my laptop this year. When deciding on the scope of what this project would entail, I was able to narrow down the full process to four key challenges, all of which were well outside of my comfort zone:

- 1.) Deploy a virtual machine.
- 2.) Install Gentoo Linux.
- 3.) Add a basic "Hello world!" system call.
- 4.) Add a `create_process()` system call that emulates the Windows `CreateProcess()` function.

### 2.2 QEMU & Gentoo Linux

When deciding on how to deploy the virtual machine, I went with QEMU since it only requires a quick install through the Linux terminal without any extensive account creation/setup that most other VM services would require. Installing Gentoo Linux involved downloading the necessary .ISO file and then loading it onto the VM to start the installation. There were a few problems installing, mainly due to the amount of space available as well as time. The laptop used in this project is over six years old, so a full compile of Gentoo required 15 hours. It failed twice before I was finally able to get it working.

### 2.3 Problems

The next step was more so a preference than a requirement, but I tried emerging Gnome to Gentoo so that I would have a user interface to work with. Unfortunately, the timeline below details roughly 245 hours of compiling Gnome, as well as its required Polkit package, that ultimately failed and set the project back about a week and a half:

- Timeline
  - Wednesday 11/20 @ 1:30p: start emerge gnome-light
  - Tuesday, 11/26 @ 5:30p: gnome crash
    - Total 148h
  - Tuesday, 11/26 @ 8:30p: resume emerge gnome-light
  - Tuesday, 11/27 @ 12a: gnome crash
    - Total 151.5h
  - Wednesday, 11/27 @ 2a: start emerge polkit
  - Wednesday, 11/27 @ 6p: pause polkit (drive to MI)
    - Total 167.5h
  - Thursday, 11/28 @ 12p: resume emerge polkit
  - Sunday, 12/1 @ 5:30p: pause polkit (drive back to IL)
  - Sunday, 12/1 @ 9:30p: polkit crash
    - Total 245h

## 2.4 Backup Plan

Since I had become annoyed with the extremely tedious and untenable Gentoo compilation issues (specifically with the optional Gnome and Polkit packages), I decided to take a risk and try it on my Ubuntu laptop with a simple “Hello world!” system call that utilizes the `printk()` function. I was going through the process simultaneously on both my host machine as well as the Gentoo VM (sans Gnome). I first had to download and decompress the Linux tarball while logged into root. Once that was complete, my *initial* understanding was that there were four main steps that needed to be followed in order to add a new system call to the kernel:

- 1.) Create a new directory (`/linux.../helloworld/`).
- 2.) In the directory, create a `helloworld.c` file as well as its corresponding Makefile.
- 3.) Add the new system call to the `syscalls.h` file.
- 4.) Assign an index for the new system call in the `syscalls_64.tbl` file.

Getting the kernel to start compiling on the host machine was relatively painless – I simply had to install a few libraries (flex, bison, and libssl-dev) and it ran smoothly thereafter. However, the Gentoo VM brought about the biggest hurdle in compiling because it involved dealing with module signatures and keys which proved problematic during the compile. I had to go into the configuration file multiple times to make adjustments. Ultimately, I ended up generating a self-signed PEM certificate, adding it into the `/certs` directory, and then pointing to it in the configuration file. The reason this was the biggest hurdle was because I was attempting a new 20-minute compile between each adjustment until it would error out. I was finally able to whittle the issue down to the PEM certificate, resolve it, and continue compiling.

Once I added my basic ‘helloworld’ syscall, I learned of another hurdle: every slight adjustment to the `syscall_64.tbl` or `syscalls.h` files would warrant a full recompile of the kernel due to the fact that any changes would propagate globally because of the critical nature of the files. Since each full recompile required a minimum of 8 hours on my laptop, this part alone became incredibly time-consuming as I kept running into undefined reference errors with ‘`sys_helloworld`’ despite following the proper implementation steps from multiple sources.

Eventually, I ended up going with a different approach that recommended getting rid of the ‘helloworld’ directory altogether, as well as its contents (‘helloworld.c’ and local ‘Makefile’), and simply adding the system call to the `fork.c` file. This side steps any linking issues since it is assumed that `fork.c` will link and compile properly. During these full recompiles, I set up a separate environment in VSCode on my desktop PC to begin working through the logic of `fork()` and coding out a very basic implementation of it in user space to be referenced later.

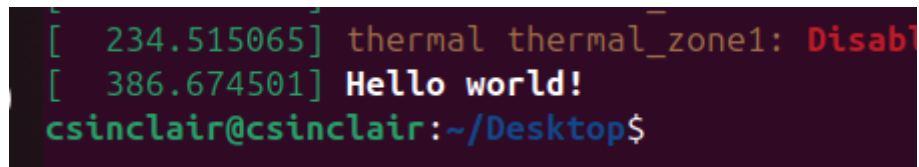
## 2.5 “Hello world!”

After several recompiles due to linking issues with the aforementioned ‘helloworld’ directory, I was finally able to get a successful compile on both the host machine and the VM when I added the syscall to `fork.c`. To mitigate any risk of potential laptop issues, I first opted to try copying the kernel image to the boot directory on the VM. Unfortunately, the GRUB configuration was not cooperating when I ran an ‘`mkconfig`’ command, so I had to manually create a duplicate of the ‘`menuentry`’ item and link it to the new image. Even then, the GRUB configuration did not initially have the root partition properly mounted. After resolving that

problem, I found that I still needed to pass the 'init' parameter into the GRUB configuration. Unfortunately, that still did not get the new image to load properly.

At this point, my patience had grown incredibly thin on the prospect of loading up the new kernel image onto the Gentoo VM within a reasonable time frame. Knowing that I had compiled successfully on both the VM and my laptop, and considering the simplicity of the system call I was adding, I took a leap of faith and decided to shift away from the VM and install the new kernel image on the laptop instead. In hopes that nothing would break, I installed the image and restarted my computer.

Upon restart, I was able to login and create a user process to test the system call. Since it uses the `printk()` function to print to the system log, I compiled the program, ran it, and then pulled up the diagnostic messages (`dmesg`) in the terminal to see if it worked...

A terminal window with a dark background. The first line shows a kernel message: [ 234.515065] thermal thermal\_zone1: Disabl. The second line shows another kernel message: [ 386.674501] Hello world!. The third line shows the shell prompt: csinclair@csinclair:~/Desktop\$.

*Alas!*

After all of the toiling to compile and recompile and recompile the kernel with every slight adjustment to key system call files, I was finally able to print "Hello world!" to the system log! I had managed to add the most basic possible system call to the Linux kernel. As basic as it was, it allowed me to confidently move on to the next, much more challenging and ambitious step: implementing the `create_process()` system call.

---

### 3. `create_process()`

As this project has been a constant learning experience, I decided that the success with implementing the above system call into Ubuntu allowed me to switch to focusing on Ubuntu for the remainder of the project. I downloaded and installed it onto a new VM altogether so as to not break my current laptop. Now knowing the process, I was able to quickly install the necessary libraries on the VM and start tinkering with the kernel code. I did run into some additional issues with the PEM certificate again, namely with trusted keys and revocation keys, but I was able to generate another self-signed PEM certificate and disable the other settings to continue.

#### 3.1 Implementation

I mentioned earlier that during the long compile times of the previous attempts to get the system call implemented, I was working out the logic of both `fork()` and `exec()` since the `CreateProcess()` call in Windows encompasses both of them. I initialized a new struct, "Process", as well as a process table array, a pointer to the current process, and a process counter. With those variables in place, I was ready to define my system call.

The beginning of `create_process()` starts with the `fork()` portion by initializing a pointer (child) to the value of the process table at the index of the process counter. Once that line of code executes, I assign the child process a process ID (PID). Now that we created a child process out of a parent process, the `fork()` portion was complete.

The `exec()` portion, however, took much more work since it entailed copying a new program into the child's memory. Since we were in kernel space, I wasn't allowed to use certain functions like `malloc()`. However, I did realize later that I could have used functions like `kmalloc()` to manage memory allocation in kernel space or `strncpy()` to copy strings, specifically to copy the "new program" (a string) into a child processes memory. I did not go that route for this attempt though – instead, I determined the program size manually by iterating through the new program (a simple string) and running a count of how many characters it contained. From there, I used pointers to iterate and set equal values between the new program and the memory of the child process. Afterwards, I null terminated the string and then set the current process to the child for the next fork.

It's important to remember that, as far as I knew, this program worked perfectly fine in user space. However, I've learned quite quickly that a lot of the norms of user space go out the window in kernel space mainly due to the extremely limited access to libraries. It becomes a very different arena and I wasn't fully confident that the function would work properly when called from user space. I was, however, confident that I had given it an earnest effort so I looked over my code once more, made my familiar updates to the `syscalls.h` and `syscall_64.tbl` files, and went for a recompile of the updated kernel.

### 3.2 Results

Unfortunately, compiling on the new Ubuntu VM gave me further trouble. The first full compile resulted in a kernel image that would not completely install due to some issues with the `AMDSCP` module. I tried turning it off in the configuration file but after multiple attempts, it still would not install. For what it's worth, I was already leery of the reliability of the compiled kernel image since the VM had crashed no less than 4 times throughout the compiling process, so I ran a 'make clean' command and started compiling a new kernel image that I could keep a closer eye on. This time, it kept crashing at the very end and terminating the VM process. I tried different approaches in launching the VM to resolve the issue, but I kept running into the ominous and nondescript 'Killed' output whenever the compile was nearly finished.

So I bit the bullet yet again and decided to try compiling a new kernel image on my host machine. After updating the kernel code, I started the final 8-hour compile and ran into a grand total of zero issues from start to finish. I'm just going to chalk it up to the laptop being too old and not powerful enough to handle memory-intensive work on virtual machines, especially after extended stretches of multiple compiling attempts.

Anyways, once the new kernel image was properly installed, I restarted the computer, crossed my fingers again in hopes that nothing would break, and was able to put together a simple test program to see if my new system call was working properly. In the program's main function, I made two `create_process()` system calls to see what would happen. Similarly to the helloworld syscall above, I had to check the diagnostic messages to see what had printed in the kernel log...

|  |   |
|--|---|
| <pre>[ 2247.785365] ath10k_pci 0000:01:00.0: [ 7] BadDELL [ 2249.418784] Hello from PID 31 [ 2249.418794] Hello from PID 30 [ 2249.418808] Hello from PID 32 [ 2249.418812] Hello from PID 31 [ 2250.971689] pcieport 0000:00:1c.0: AER: Cor</pre> | <pre>csincaltr@csincaltr: ~/Desktop\$ ./createProcess Hello from PID 1 Hello from PID 0 Hello from PID 2 Hello from PID 1 Final process (PID 2): New Program Executed</pre> |
|--|---|

*Kernel log output*

*User space test program output*

This is definitely a promising result, but when placed alongside the user space test program that I wrote during the kernel compiling process, something doesn't line up. The final output line to confirm proper execution of the `exec()` portion of the code is missing from the kernel log. This shows that I was able to implement the `fork()` portion properly, but the `exec()` portion's output is still missing due to the way that I was testing the syscall. The main issue was that the final `printf()` output was not present in the kernel implementation. I was mistakenly trying to call the kernel space values of the current processes PID and memory content from user space – truly an amateur move.

Thankfully, I did not have to update the `syscalls.h` or `syscall_64.tbl` files at all, so a full recompile of the kernel was not necessary. I made a quick update to `kernel/fork.c` that adjusted the program so that it would anticipate 2 system calls from user space so that the process that ran the `exec()` function (the deepest child process), would print the final line of output into the kernel log:

|   |  |
|---|--|
| <pre>[ 35.556407] Hello from PID 1 [ 35.556437] Hello from PID 2 [ 35.556442] Hello from PID 1 [ 35.556445] Final process (PID 2): New Program Executed</pre> | <pre>csincaltr@csincaltr: ~/Desktop\$ ./createProcess Hello from PID 1 Hello from PID 2 Hello from PID 1 Final process (PID 2): New Program Executed</pre> |
|---|--|

*Kernel log output*

*User space test program output*

*Partial success!*

I finally managed to 1.) *cover the `fork()` portion by creating a child process out of a parent process*, and 2.) *implement the `exec()` portion by properly executing a new program in the deepest child process*. Unfortunately, as you can see in the images above, I did manage to lose the line of output from PID 0. In trying to diagnose the issue, I noticed that in user space, initializing the process count integer to 0 was reflected as such in the printed output upon entering the method. However, even after setting the process count to 0 as a static integer in kernel space, for some reason it was still outputting as 2 upon entering the system call in the kernel log. I think it's safe to say that this is a work on progress.

Either way, I would argue that this qualifies as a *very basic* implementation of `create_process()` as it still manages to cover the essentials. However, it's worth noting that the actual `fork.c` file contains roughly 3,500 lines of code, so my implementation is far from accurate. The actual output of two consecutive `fork` calls would contain four unique PIDs, with an output order that depends on the underlying architecture. My understanding though is that the deepest leaf node of the resulting tree would be the first child process to execute. My implementation ignores that established order entirely and simply creates a child process out of its parent process without regard to a larger data structure, such as a tree. It then executes a new function into the deepest child process, printing the final line of output in the kernel log.

## 4. Conclusion

In conclusion, my main question is: how on earth do people develop at the kernel level when it requires an insane amount of compile time (as well as install time) just to test new kernel functions? Even on newer equipment, compiling the kernel would still take at least an hour. Regardless, this project was definitely a massive step outside of my comfort zone and the amount of things I learned probably warrants a brief list:

- How to deploy a QEMU VM.
- How to install Linux (Gentoo & Ubuntu).
  - Resizing the disk image and remounting the root partition (many times).
  - Installing necessary libraries for compiling.
- How to modify kernel code at the root level to implement a system call.
  - Modifying configuration files.
  - Generating a PEM certificate.
- How `CreateProcess()` works on the surface level by combining `fork()` and `exec()` into one function.
  - Emulating a basic synthesis of that function within kernel space.
  - Testing syscall logic in user space and adjusting it for kernel implementation.
  - Confirming proper kernel space output via the kernel log.

With that, I also feel that it's important to give a shoutout to my 6-year old laptop for barreling through the last month of work almost 24/7 without breaking down completely (though it did get close). I considered this thing a paperweight up until September, but it has been my saving grace throughout the semester, allowing me to learn so much more than I would have without it, especially when it comes to Linux and C. I think it's safe to say that I learned a whole lot about operating systems in these courses, ultimately culminating in this deep dive into the process of changing the very code of an OS. This is a project that, if time permits, I would definitely like to continue exploring.